Драйверы LINUX

Общие понятия

- Строго говоря, драйвером считается фрагмент кода операционной системы, который позволяет ей обращаться к аппаратуре. Не вполне конкретный термин "аппаратура" обозначает здесь как неотъемлемые части компьютера (например, наборы микросхем на материнских платах современных персональных компьютеров), так и вполне автономные устройства (как, скажем, "древние" устройства считывания с перфокарт, редко размещавшиеся в одной комнате с процессорной стойкой).
- Концепция драйвера как отдельного сменного модуля оформилась не сразу. Некоторые версии UNIX и по сию пору практикуют полную перекомпиляцию ядра при замене какого-либо драйвера, что совершенно не похоже на обращение с драйверами в Linux, Windows и MS DOS. Кстати, именно MS DOS ввела в массовое обращение понятие драйвера, как легко сменяемой насадки, позволяющей моментально (сразу после очередной перезагрузки) улучшить качество жизни пользователя

Общие понятия

- Касаясь характерных черт драйвера (работающего с полномочиями компонента ядра) для разных операционных систем - именно, Windows и Linux - остановимся на трех неслучайных совпадениях.
- **Наблюдение 1.** В операционных системах MS DOS, Windows, Unix и всех клонах Linux принят способ работы с драйверами как с файлами. То есть при доступе к драйверу используются функции либо совпадающие (лексически), либо весьма похожие на функции для работы с файлами (**open, close, read, write, CreateFile**...).
- Данный порядок неудивителен для систем юниксоидного ряда, поскольку в них вся действительность воспринимается в виде файлов (что является изначальной концепцией данной ветви операционных систем). Например, директорию (каталог файлов) можно открыть как файл и считывать оттуда блоки данных, соответствующие информации о каждом хранящемся в этой директории файле.

Общие понятия

- В директории /dev/ можно открыть файл, соответствующий мышке и считывать постепенно байты данных, появляющиеся в нем в точном соответствии с ее перемещениями.
- В Windows предлагается точно такой же механизм. Для доступа к драйверу из своего приложение пользователь прибегает к помощи функции **CreateFile**. Правда, имя файла, который предполагается "открыть", выглядит странно, как "\\\\.\\myDevice". (Операционная система понимает его как символьную ссылку для идентификации конкретного драйвера, привлекаемого к работе.) И хотя дальнейшие операции, сформулированные создателем пользовательского приложения как вызовы **read**()-**write**(), все-таки преобразуются операционной системой в специальные запросы к драйверу, необходимо признать: формально процесс похож на работу с файлом.

Общие понятия

- **Наблюдение 2**. Драйверы стали легко заменяемой запасной частью в операционной системе. Если раньше и были различия между продуктами Microsoft и юниксоидными системами (драйверы в операционных системах Microsoft изначально были "подвижно-сменными", но в UNIX и ранних версиях Linux при их замене надо было заново выполнять перекомпиляцию ядра), то сейчас такие различия исчезли. При сохранении некоторых особенностей инсталляции, драйверы теперь повсеместно могут быть удалены/добавлены в систему редактированием одной записи в специальных системных файлах. Более того, загрузка "по требованию" (по запросу пользовательской программы) становится практически общей чертой Windows/Unix/Linux. Даже операционные системы реального времени, например, QNX также используют методику сменных драйверов.

Общие понятия

- **Наблюдение 3.** Концепция существования режима ядра (с большими функциональными возможностями и относительной бесконтрольности) и пользовательского режима (с жестким контролем со стороны системы) присутствует в Windows/Unix/Linux с незапамятных времен. Если внимательно посмотреть на то, как в Linux реализуется драйвер, то увидим, что это всего лишь модуль ядра, который имеет некое (дополнительное) отражение в виде файла в директории /dev/. Если посмотреть теперь на драйвер (режима ядра) в операционной системе Windows, то становится понятно: это не просто драйвер, это возможность войти в режим ядра со своим программным кодом.
- Завершая мини-экскурс в сравнительный анализ драйверов разных популярных ОС, нельзя не упомянуть и об общем для всех систем механизме воздействия на драйвер при помощи IOCTL запросов.

Общие понятия

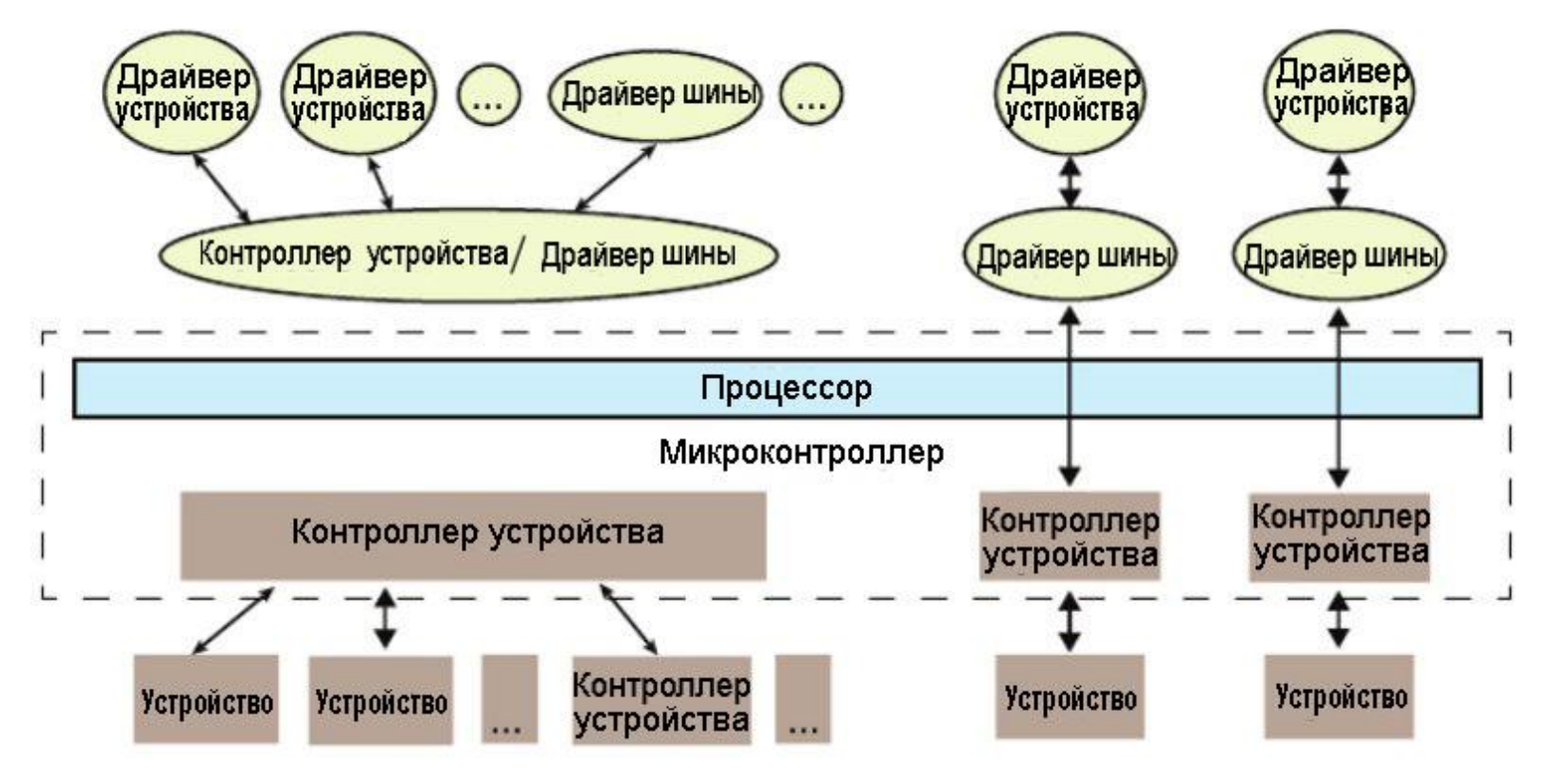
- Итак, драйвер управляет, контролирует, следит за работой объекта, который подчиняется командам драйвера. Драйвер шины управляет работой шины, драйвер устройства управляет работой устройства (частью оборудования, подключенного к компьютеру), например, мышью, клавиатурой, монитором, жестким диском и многим другим. Управление определенной частью аппаратных средств может осуществляться некоторой частью программного обеспечения (драйвером устройства), или может осуществляться другим устройством, управление которым, в свою очередь, может выполняться программой — драйвером устройства. В последнем случае, такое управляющее устройство обычно называется контроллером устройств. Для него, поскольку оно само является устройством, необходим драйвер, который обычно называют bus driver или драйвер шины.

Общие понятия

- К числу примеров контроллеров устройств относятся контроллеры жестких дисков, контроллеры дисплеев и контроллеры аудиоустройств, с помощью которых осуществляется управление устройствами, подключаемыми к контроллерам. В качестве более подробных технических примеров можно рассмотреть контроллер IDE, контроллер PCI, USB-контроллер, контроллер SPI, контроллер I2C и т.д. Графически, вся эта концепция может быть изображена так, как показано на рис.1.
- Контроллеры устройств, как правило, подключаются к процессору через шины, имеющие определенное название (набор физических линий подключения) - например, шина PCI, шина IDE, и т.д. В современном мире встроенных технологий мы чаще сталкиваемся с микроконтроллерами, а не процессорами; это те же самые процессоры и плюс контроллеры различных устройств, реализованные на одном чипе. В таких случаях шины интегрированы в сам чип. Меняет ли это что-либо для драйверов или, в более общем случае, в используемом программном обеспечении?

Общие понятия

Рис.1: Взаимодействие устройств и драйверов



Общие понятия

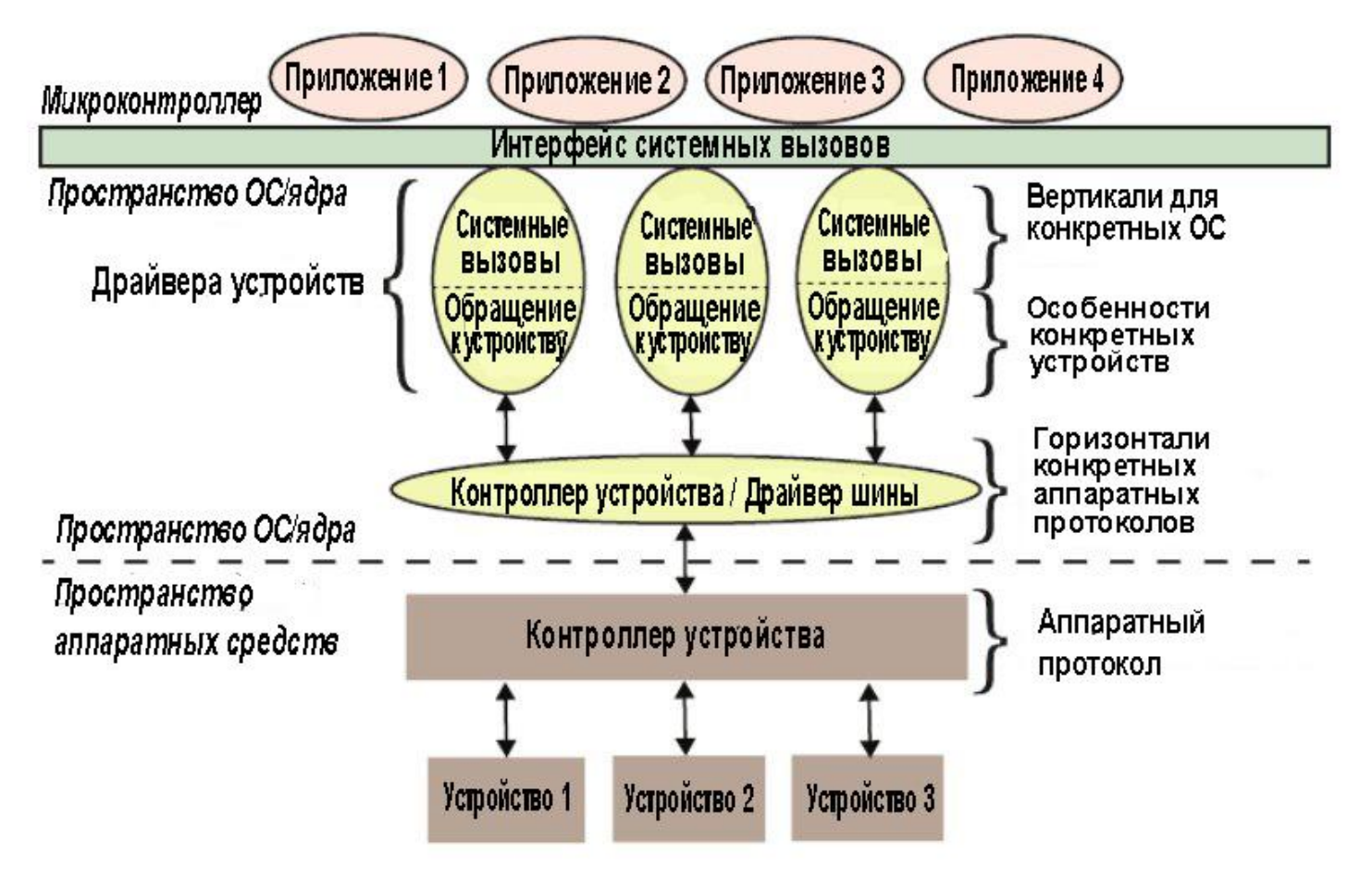
- Ответ на этот вопрос не так уж сложен — разве что драйверы шин для контроллеров соответствующих встроенных устройств будут теперь разрабатываться под зонтиком конкретной архитектуры.
- ***Драйверы состоят из двух частей***
- В драйверах шин предоставляются специальные аппаратные интерфейсы для соответствующих аппаратных протоколов оборудования и эти драйверы являются самыми нижними горизонтальными программно реализуемыми слоями операционной системы (ОС). Над ними расположены драйверы конкретных устройств. Они работают с лежащими ниже устройствами через горизонтальный слой интерфейсов и разрабатываются для каждого конкретного устройства. Тем не менее, сама идея написания таких драйверов позволяет предоставить пользователю абстрагированный доступ и, тем самым, реализовать на другом "конце" интерфейс (который будет варьироваться в зависимости от ОС).

Общие понятия

- Короче говоря, драйвер устройства состоит из двух частей, одна из которых а) является специфической для конкретного устройства, а другая б) является специфической для ОС. Смотрите рис.2.
- Часть драйвера устройства, характерная для конкретного устройства, будет одной и той же во всех операционных системах и в большей мере она связана с анализом и пониманием спецификаций устройства, а не с программированием. Спецификации устройства представляют собой документ, в котором описываются технические особенности устройства, в том числе его функционирование, пропускную способность, программирование и т.д. - в общем, это - руководство пользователя устройства.

Общие понятия

Рис.2: Отдельные части драйвера Linux



Общие понятия

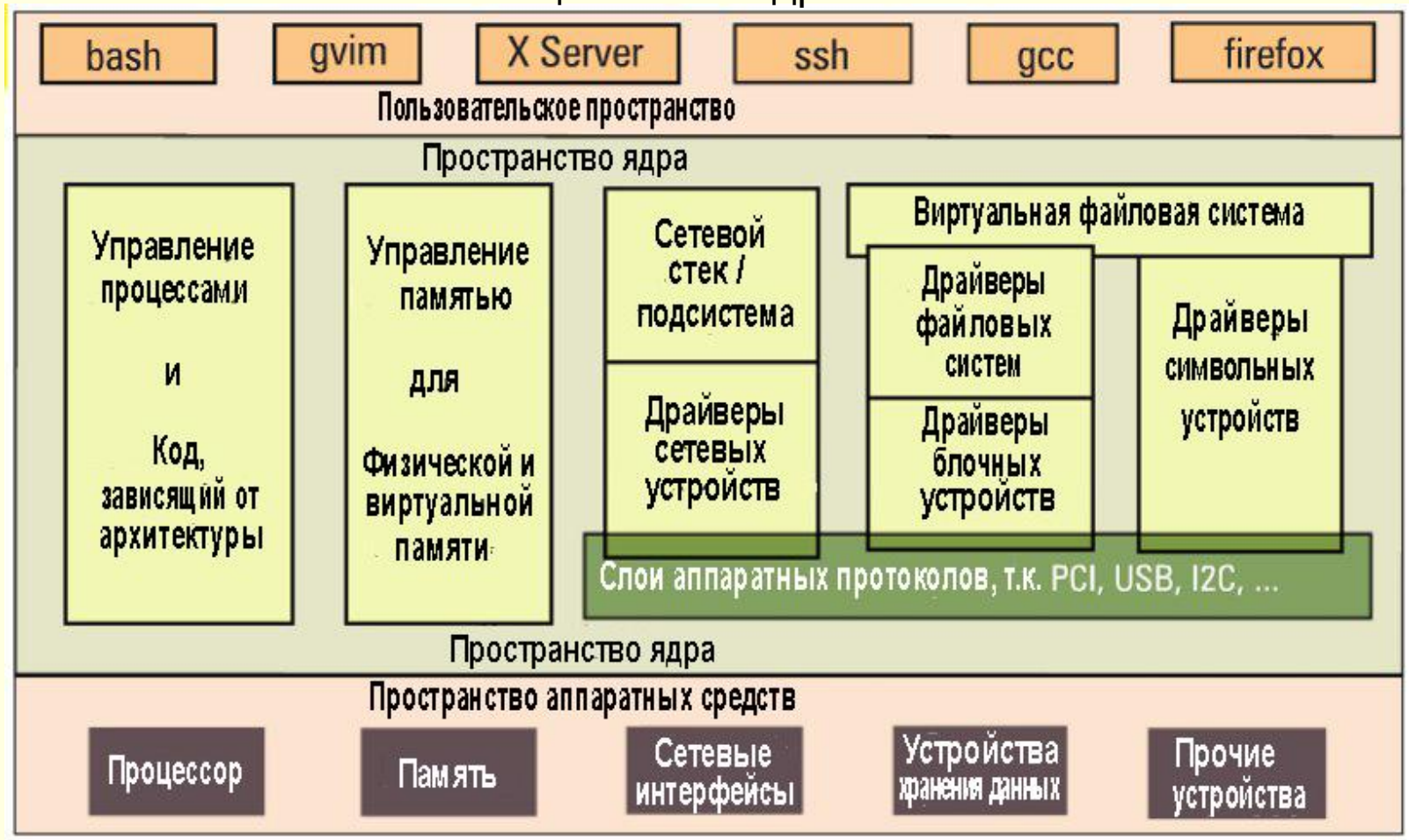
- Та часть драйвера, которая зависит от ОС, тесно взаимодействует с механизмами ОС, реализующими пользовательский интерфейс, и, поэтому, она будет отличаться в драйверах устройств для Linux, в драйверах устройств для Windows и в драйверах устройств для MacOS.
- ***Вертикали***
- В Linux драйвер устройства предоставляет пользователю интерфейс "системного вызова"; в Linux это граница между так называемым пространством ядра и пользовательским пространством, что и показано на рис.2. На рис.3 представлена более подробная классификации.
- Если рассматривать интерфейс драйвера с учетом специфики использования драйверов в ОС, то в системе Linux драйверы можно по вертикали грубо разделить на три группы:

Общие понятия

- Пакетно-ориентированная или сетевая вертикаль
- Блочно-ориентированная вертикаль или вертикаль хранения данных
- Байт-ориентированная вертикаль или вертикаль работы с символами
- Вертикаль процессора и вертикаль памяти, рассматриваемые вместе с этим тремя вертикалями, дают полное представление о ядре Linux, соответствующее определению ОС, которое есть в любом учебнике: "В операционной системе реализуется 5 основных функций управления: управление процессором / процессом, памятью, сетью, средствами хранения данных, устройствами ввода / вывода". Хотя эти вертикали процессора и памяти можно классифицировать как драйверы устройств, где процессор и память будут соответствующими устройствами, их, по ряду причин, трактуют по-другому.

Общие понятия

Рис.3: Общая схема ядра Linux



Общие понятия

- Таковы основные функциональные возможности любой ОС, будь то микроядро или монолитное ядро. Чаще всего добавление кода именно в эти области представляет собой основную часть работы по портированию Linux, что обычно делается для нового процессора и архитектуры. Более того, код в этих двух вертикалях нельзя, в отличие от трех других вертикалей, загружать или выгружать "на лету". Так что когда мы теперь будем говорить о драйверах устройств в Linux, мы будем говорить только о тех трех вертикалях, которые расположены на рис.3 справа.
- Давайте заглянем глубже внутрь этих трех вертикалей. Сетевая вертикаль состоит из двух частей: а) стек сетевых протоколов и б) драйверы устройств карт сетевых интерфейсов (NIC) или просто драйверы сетевых устройств, которые могут предназначаться для Ethernet, Wi-Fi или любой другой сетевой горизонтали. Вертикаль хранения данных, опять же, состоит из двух частей:

Общие понятия

- а) драйверов файловых систем, предназначенных для декодирования разнообразных форматов данных в различных разделах файловых систем, и
- б) драйверов блочных устройств для различных (аппаратных) протоколов хранения данных, т.е. горизонталей, таких как IDE, SCSI и т.д.
- В действительности из-за огромного количества драйверов в этой вертикали, для драйверов, предназначенных для работы с символьными устройствами, используется дополнительная подклассификация - так что у вас есть драйверы терминалов, драйверы ввода/вывода, драйверы консоли, драйверы фрейм-буфера, звуковые драйверы и т.д. Типичными горизонталями здесь будут RS232, PS/2, VGA и т.д.

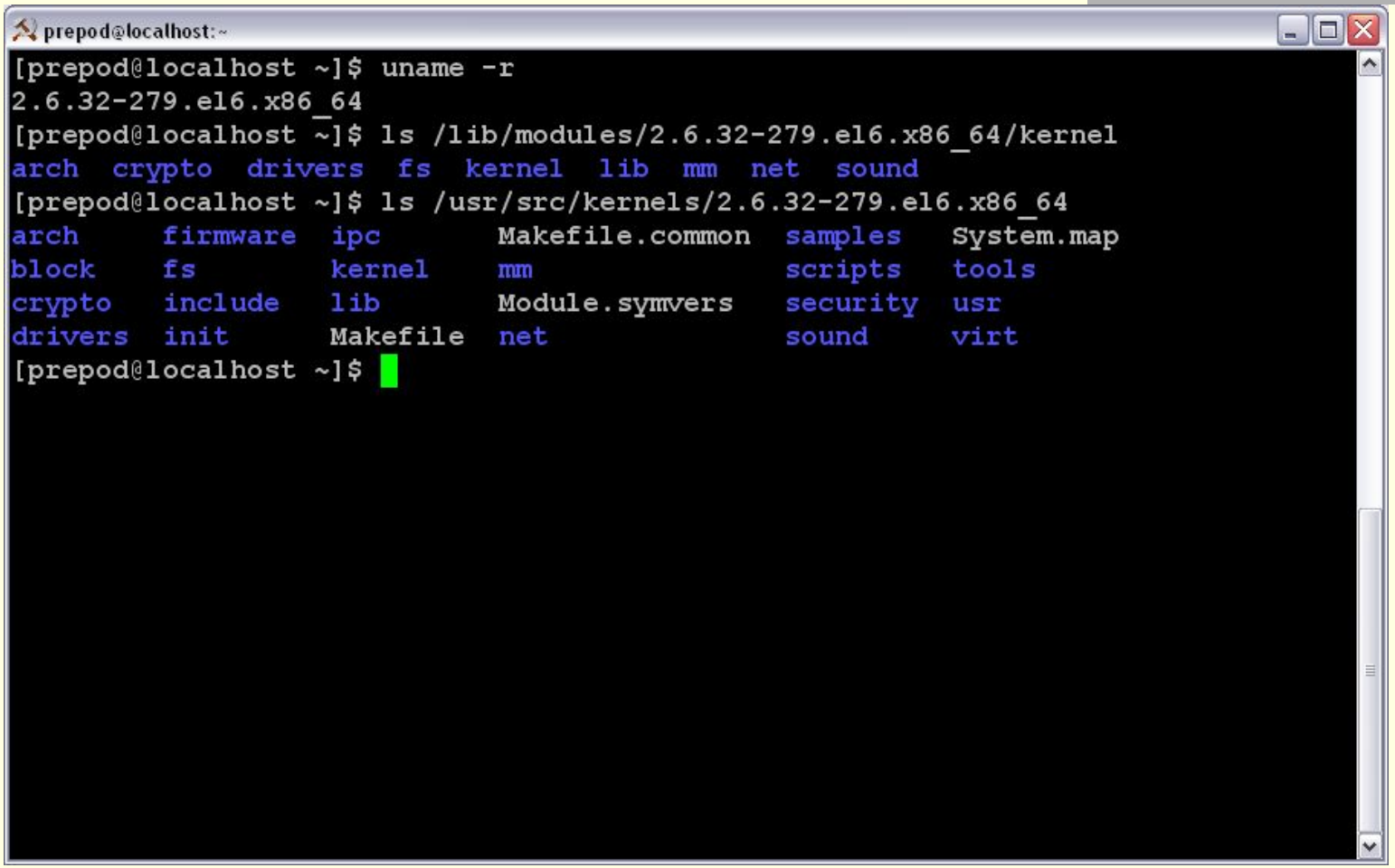
Общие понятия

- ***Драйверы со множественными вертикалями***
- Последнее замечание относительно полной картины (размещения всех драйверов в экосистеме драйверов Linux): такие горизонтали, как USB, PCI и т.д., расширяются ниже на несколько вертикалей. Может быть USB Wi-Fi, флешка USB и преобразователь USB-последовательный порт, но все эти устройства USB попадают в три различные вертикали!
- В Linux драйвера шин или горизонтали часто подразделяются на две части, или даже на два драйвера: а) контроллер конкретного устройства и б) абстрактный слой, находящийся над ним и используемый в качестве интерфейсам к вертикалям, обычно называемыми ядрами. Классическим примером могут быть драйвера USB-контроллера ohci, ehci и т.д. и USB-абстракция - usbcore.

Динамическая загрузка драйверов

- Динамически загружаемые драйвера чаще всего называют модулями, которые собираются в виде отдельных модулей с расширением .ko (объект ядра). В каждой системе Linux в корне файловой системы (/) есть стандартное место для всех предварительно собранных модулей. Они организованы аналогично древовидной структуре исходных кодов ядра и находятся в директории /lib/modules/<kernel_version>/kernel, где <kernel_version> результат вывода системной команды uname -r (см.рис.4).

Динамическая загрузка драйверов



Динамическая загрузка драйверов

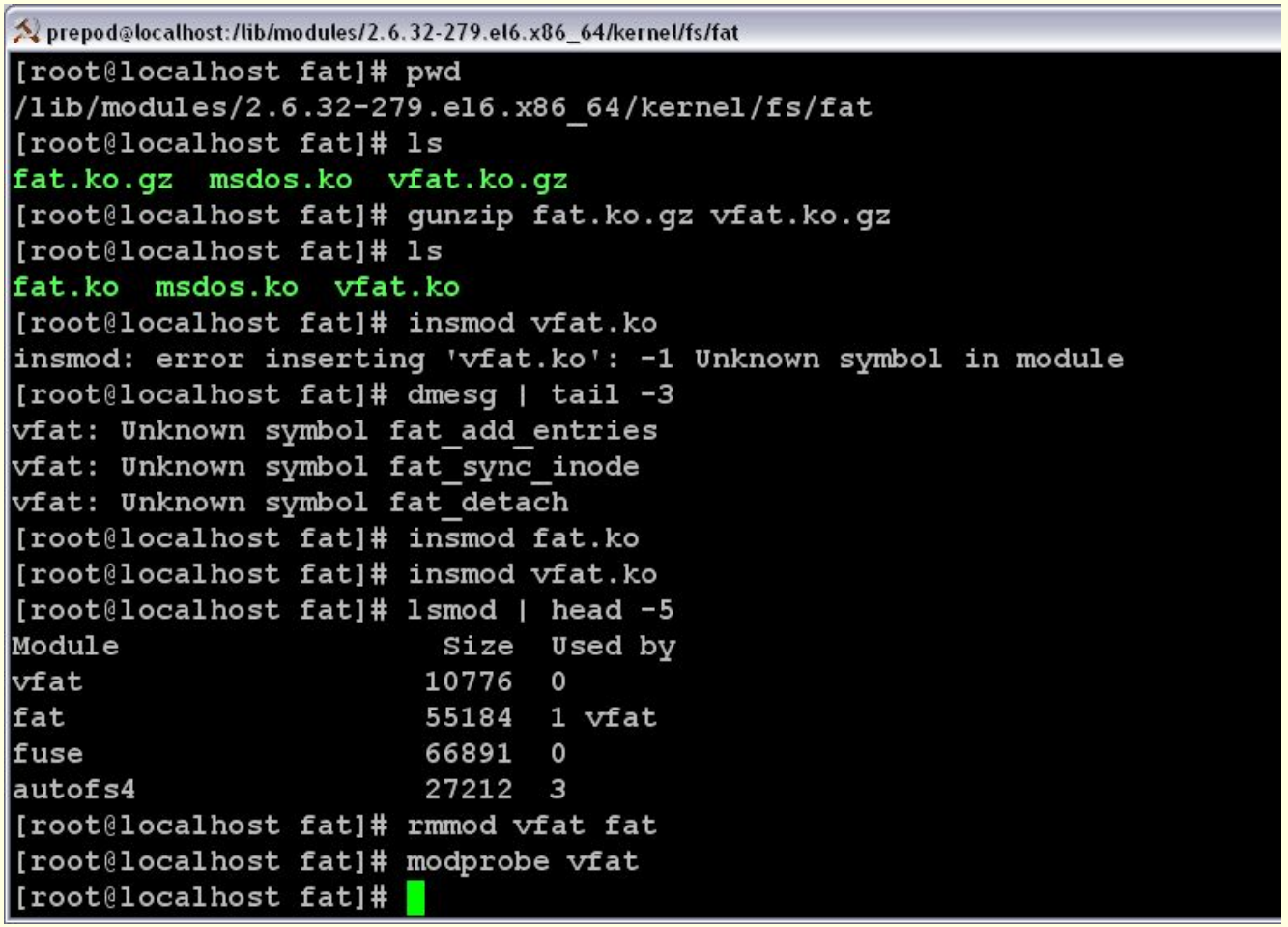
- Чтобы динамически загружать и выгружать драйверы, воспользуйтесь следующими командами, которые находятся в директории /sbin и должны выполняться с привилегиями пользователя root:
  - **lsmod** — список модулей, загруженных в текущий момент
  - **insmod** <module_file> — добавление / загрузка указанного файла модуля
  - **modprobe** <module> — добавление / загрузка модуля вместе со всеми его зависимостями
  - **modinfo** <module> — получение информации о модуле
  - **rmmod** <module> — удаление / выгрузка модуля

Динамическая загрузка драйверов

- Давайте в качестве примера рассмотрим соответствующие драйвера файловой системы FAT. На рис.5 показан весь процесс нашего эксперимента. Файлы с модулями будут fat.ko, vfat.ko и т.д., находящиеся в директории fat (в vfat для старых версий ядра) в /lib/modules/`uname -r`/kernel/fs. Если они представлены в сжатом формате .gz, вам нужно будет распаковать их с помощью команды gunzip, прежде чем вы сможете выполнить операцию insmod.
- Модуль vfat зависит от модуля fat, так что первым должен быть загружен модуль fat.ko. Чтобы автоматически выполнить распаковку и загрузку зависимостей, воспользуйтесь командой modprobe. Обратите внимание, что когда вы пользуетесь командой modprobe, вы не должны в имени модуля указывать расширение .ko. Команда rmmod используется для выгрузки модулей.

Динамическая загрузка драйверов

Рис.5: Операции с модулями Linux



Динамическая загрузка драйверов

- Здесь также вызывается команда **dmesg** (сокр. от англ. *display message* или англ. *driver message*) — команда, используемая в UNIX-подобных операционных системах для вывода буфера сообщений ядра в стандартный поток вывода (stdout) (по умолчанию на экран). Буфер содержит все сообщения ядра, начиная со времени загрузки ОС. Для проверки последних событий в ОС вывод команды «dmesg» перенаправляют на фильтр «tail», чтобы отфильтровать только последние 3 строки буфера:
- Вывод lsmod перенаправляется на фильтр head для вывода первых 5 строк.

Первый драйвер для Linux

- Драйвер никогда не работает сам по себе. Он похож на библиотеку, загружаемую из-за функций, которые будут вызваны из работающего приложения. Он написан на языке С, но в нем отсутствует функция main(). Кроме того, он будет загружаться / компоноваться с ядром, поэтому он должен компилироваться аналогично тому, как было откомпилировано ядро, и вы можете в качестве заголовочных файлов использовать только те, что есть в исходном коде ядра, а не из стандартного директория /usr/include.
- Интересный факт, касающийся ядра, это то, что оно представляет собой объектно-ориентированную реализацию на языке С. В любом драйвере есть конструктор и деструктор. Когда модуль успешно загружается в ядро, то вызывается конструктор модуля, а дескруктор модуля вызывается, когда команде rmmod удается успешно выгрузить модуль.

Первый драйвер для Linux

- Это в драйвере две обычные функции, разве что они называются *init* и *exit*, соответственно, и вызываются с помощью макросов module_init() и module_exit(), которые определены в заголовков ядра module.h.
- С учетом вышесказанного это полный код нашего первого драйвера; назовем его mfd.c. Обратите внимание, что отсутствует заголовок stdio.h (заголовок пользовательского пространства), вместо него мы используем аналог kernel.h (заголовок пространства ядра). Функция printk() эквивалентна функции printf(). Разница лишь в том, что при программировании ядра, нам не потребуется беспокоиться о плавающих форматах %f, %lf и тому подобном. Но, в отличие от команды printf, команда printk не предназначена для выдачи дампа своих данных в какую-нибудь консоль. Кроме того, для обеспечения совместимости версии модуля с ядром, в которое будет загружен модуль, добавлен заголовок version.h. С помощью макроса MODULE_* заполняется информация, относящаяся к модулю, которая будет использована как "подпись" модуля.

Сборка первого драйвера

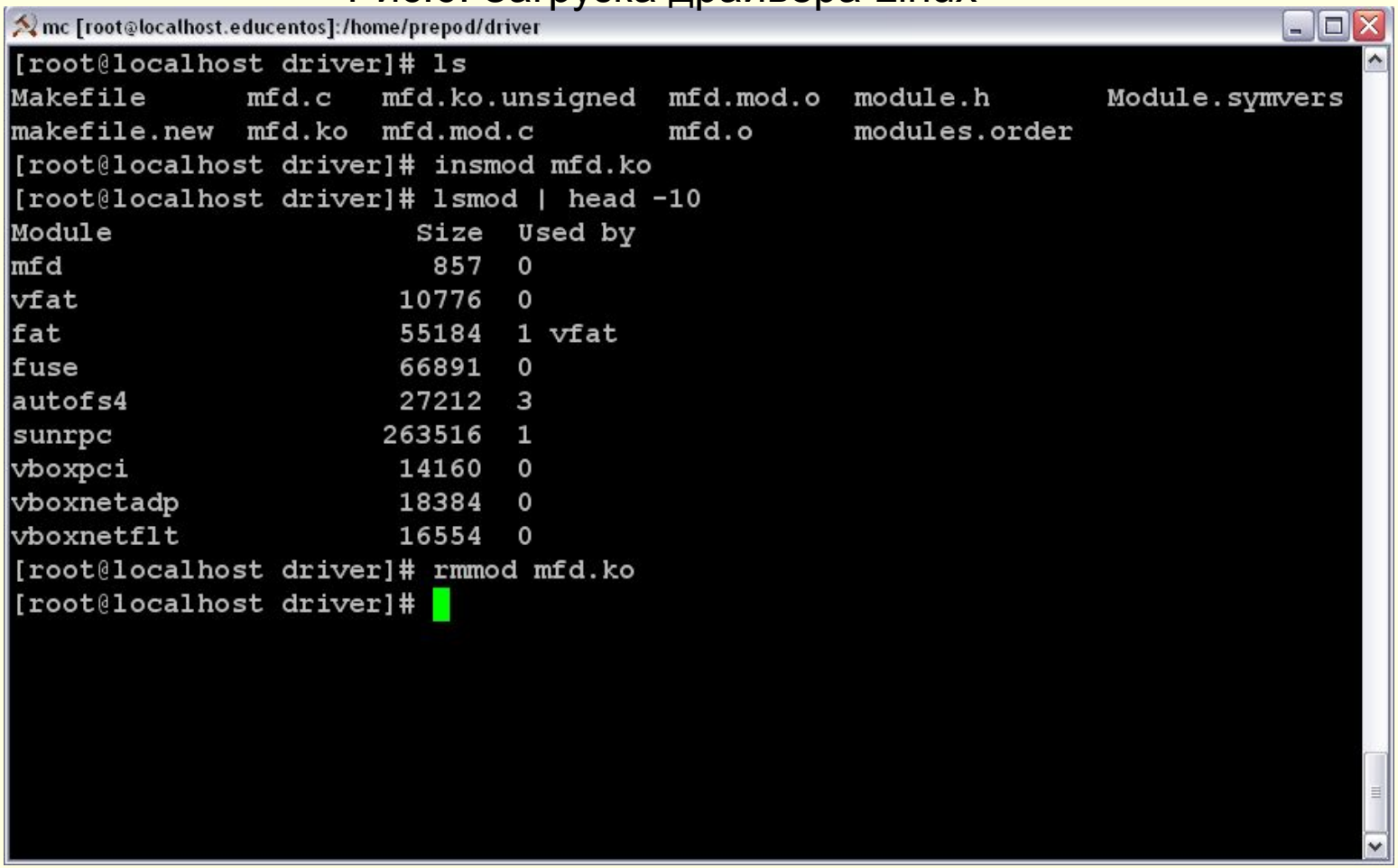
- Так как у нас есть код на языке C, настало время его скомпилировать и создать файл модуля mfd.ko. Для этого мы используем систему сборки ядра. В приведенном ниже файле Makefile происходит обращение к системе сборки ядра из исходных кодов, а файл Makefile ядра, в свою очередь, обращается к файлу Makefile нашего нового драйвера с тем, чтобы собрать драйвер.
- Чтобы собрать драйвер для Linux, у вас в системе должен быть исходный код ядра (или, по крайней мере, заголовки ядра). Предполагается, что исходный код ядра будет находиться в директории /usr/src/linux. Если в вашей системе он находится в каком-нибудь другом месте, то укажите это место в переменной KERNEL_SOURCE в файле Makefile.
- Если исходные коды отсутствуют, их необходимо установить, например: rpm -Uhv ftp://rpmfind.net/linux/centos/6.3/centosplus/x86_64/Packages/kernel-2.6.32-279.1.1.el6.centos.plus.x86_64.rpm

Сборка первого драйвера

- Версию ядра можно определить утилитой uname –a
- Найти пакет можно на http://rpmfind.net/linux/rpm2html/search.php?query=kernel
- Можно устанавливать не все исходные коды, а только заголовки с помощью утилиты yum: **`yum install kernel-devel`**
- Содержимое Makefile лежит здесь. Расширения у него не должно быть, а имя – регистрозависимое.
- Когда есть код на языке C (mfd.c) и готов файл Makefile, то все, что нам нужно сделать для сборки нашего первого драйвера (mfd.ko), это вызвать команду make.
- **`# make -C /usr/src/linux SUBDIRS=$PWD modules`**
- Как только у нас будет файл mfd.ko, мы в роли пользователя root выполним обычные действия.
- **`# su`**
- **`# insmod mfd.ko`**
- **`# lsmod | head -10`**
- Команда lsmod должна вам сообщить о том, что драйвер mfd загружен (Рис. 6):

Сборка первого драйвера

Рис.6: Загрузка драйвера Linux



Журнал сообщений ядра

- Команда printk не предназначена для выдачи дампа своих данных в какую-нибудь консоль.
- На самом деле, она не может это делать; это нечто, что сидит в фоновом режиме и выполняется точно также, как библиотека, только тогда, когда она запускается либо из пространства аппаратных средств, либо из пространства пользователя. Все вызовы команды printk помещают свои выходные данные в кольцевой буфер (журнал) ядра. Затем демон syslog, работающий в пользовательском пространстве, берет их для окончательной обработки и перенаправляет на различные устройства в соответствие с тем, что задано в файле конфигурации /etc/syslog.conf.
- В вызовах printk вы должны были обратить внимание на макрос KERN_INFO. Это, в действительности, строковая константа, которая объединяется в одну строку со строкой формата, идущей за ней. Обратите внимание, что между ними нет запятой (,), это не два отдельных аргумента. В исходном коде ядра есть восемь таких макросов, которые определены в linux/kernel.h, а именно:

Журнал сообщений ядра

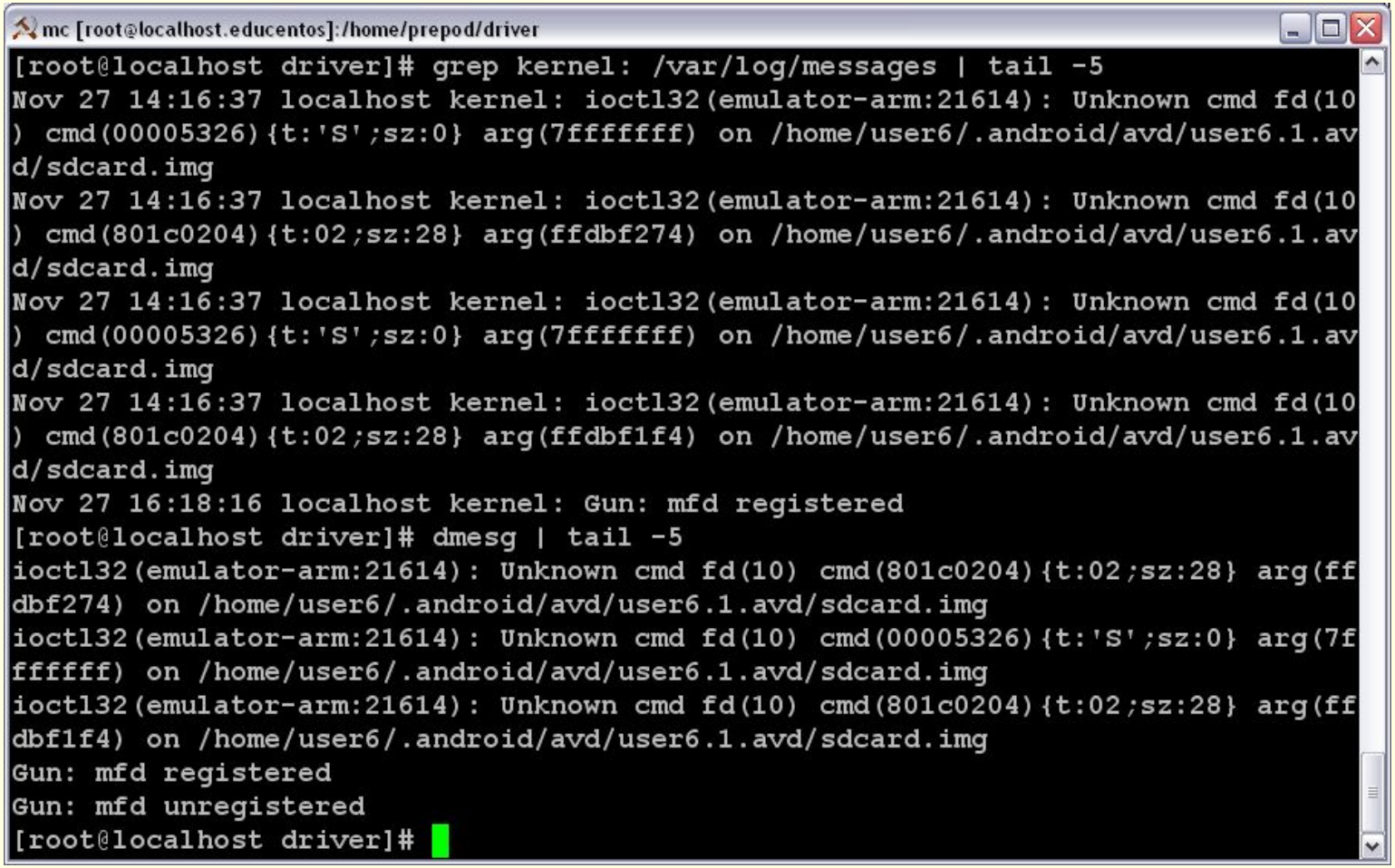
- `#define KERN_EMERG "<0>"  /* system is unusable */`
- `#define KERN_ALERT "<1>"/*action must taken immediately*/`
- `#define KERN_CRIT "<2>"    /* critical conditions */`
- `#define KERN_ERR "<3>"    /* error conditions */`
- `#define KERN_WARNING "<4>" /* warning conditions */`
- `#define KERN_NOTICE "<5>"/*normal significant condition*/`
- `#define KERN_INFO "<6>"    /* informational */`
- `#define KERN_DEBUG "<7>"  /* debug-level messages*/`
- Теперь, в зависимости от этих уровней журналирования (то есть первых трех символов в строке формата), демон пользовательского пространства **`syslog`** перенаправляет каждое сообщения в соответствие с заданной конфигурацией. Обычно местом, куда перенаправлются сообщения всех уровней журналирования, является журнальный файл **`/var/log/messages`**.

Журнал сообщений ядра

- Таким образом, все данные, выдаваемые командой printk, по умолчанию находятся в этом файле. Впрочем, можно изменить настройку — например, пересылать сообщения на последовательный порт (например, /dev/ttyS0) или на все консоли, как это обычно происходит в случае возникновения события KERN_EMERG.
- Сообщения теперь находятся в буфере **`/var/log/messages`**, причем в нем находятся сообщения не только из ядра, но и от различных демонов, работающих в пользовательском пространстве. К тому же, этот файл обычно нельзя читать от имени обычного пользователя. Поэтому для непосредственного разбора сообщений, находящихся в кольцевом буфере ядра, предоставляется утилита пользовательского пространства dmesg, которая выводит дамп буфера в стандартный выходной поток. На рис.7 показаны фрагменты вывода в стандартный выходной поток.

Журнал сообщений ядра

Рис.7: Журналирование сообщений ядра



Язык С ядра — чистый С

- Для любой функции ядра требуется обработка ошибок, как правило, возвращаемых в виде целочисленного типа, причем возвращаемое значение должно соответствовать следующему правилу. При ошибке мы возвращаем отрицательное число: минус добавляется макросом, находящимся в заголовке ядра Linux linux/errno.h, который включает в себя заголовки различных ошибок в исходном коде ядра, а именно - asm/errno.h, asm-generic/errno.h, asm-generic/errno-base.h.
- При успешном завершении в случае, когда не должна предоставляться некоторая дополнительная информация, наиболее распространенным возвращаемым значением будет ноль. В случае, когда возвращается положительное значение, то оно будет указывать дополнительную информацию, например, количество байтов, возвращаемых функцией.

Язык С ядра — чистый С

- Стандартный язык С является только языком программирования. Заголовочные файлы не являются его частью. Это часть стандартных библиотек, собранных для программистов на языке С и реализующих концепцию повторного использования кода.
- Разработчики ядра разработали свой собственный набор необходимых функций, которые являются частью кода ядра. Функция printk является лишь одной из них. Аналогичным образом многие функции, предназначенные для работы со строками, функции работы с памятью и многое другое, являются частью исходного кода ядра; они расположены различных директориях kernel, ipc, lib и так далее, вместе с соответствующими заголовочными файлами, которые находятся в директории include/linux.

Символьные драйверы Linux

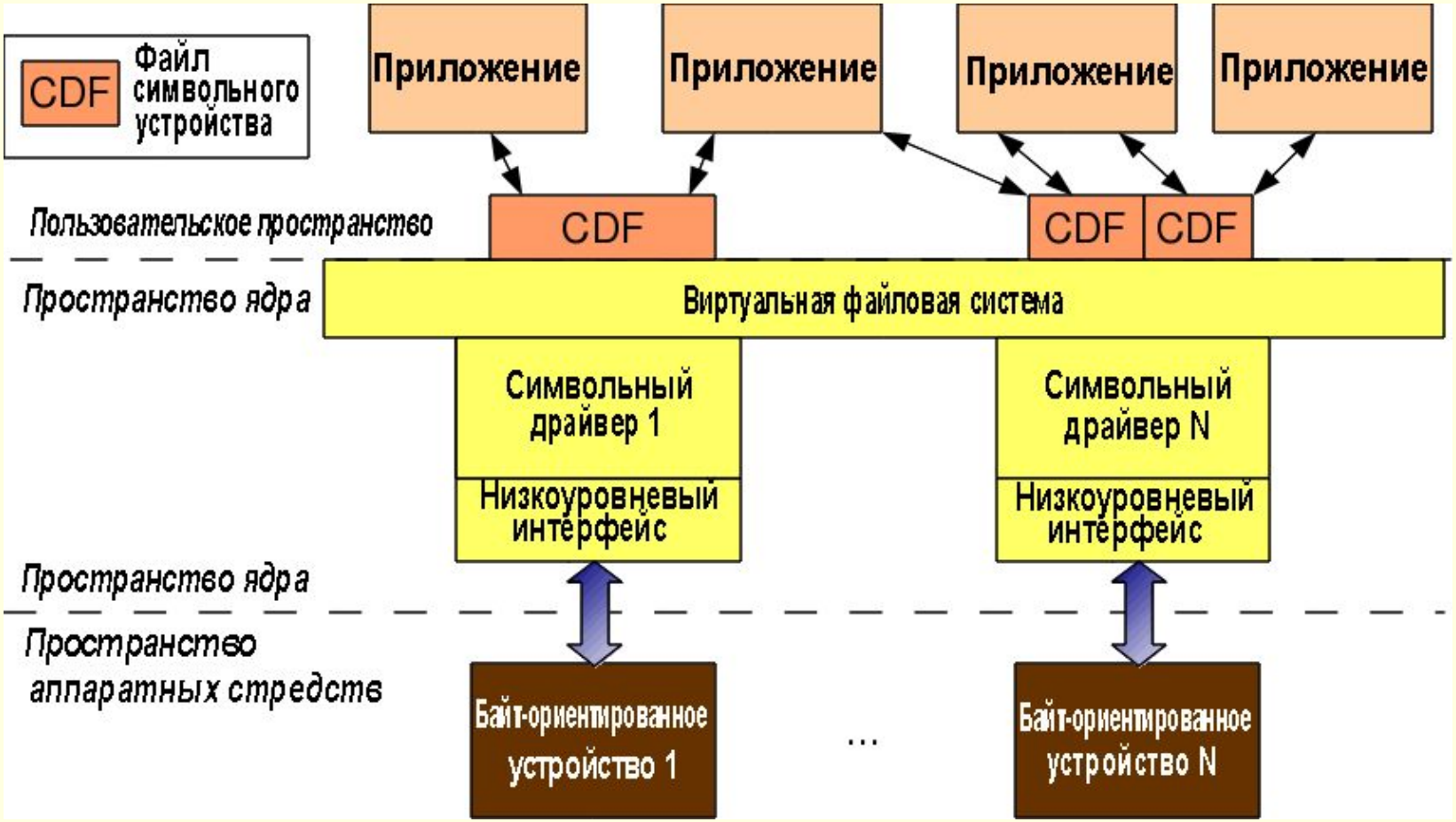
- Нереально за несколько лекций пересказать книгу "Драйверы устройств Linux" Джонатана Корбета, Алессандро Рубини и Грега Кроа-Хартмана (*Linux Device Drivers* Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman). Русского перевода 3го издания нет, есть второй.
- И тем не менее
- ***Все о символьных драйверах***
- Если мы пишем драйверы для байт-ориентированных операций (или, на жаргоне языка C, символьно-ориентированных операций), то мы называем их символьными драйверами. Поскольку большинство устройств является байт-ориентированными, то большинство драйверов устройств являются символьными драйверами.

Символьные драйверы Linux

- Возьмем, к примеру, драйверы последовательного порта, аудио драйверы, и драйверы базового ввода/вывода. На самом деле, все драйверы устройств, которые не являются ни драйверами устройств хранения данных, ни драйверами сетевых устройств, будут символьными драйверами некоторого вида. Давайте рассмотрим общие особенности этих символьных драйверов.
- Как показано на рис.8, для любого приложения пользовательского пространства, предназначенного для работы с байт-ориентированным устройством (в пространстве аппаратных средств), следует использовать соответствующий драйвер символьного устройства (в пространстве ядра). Использование символьных драйверов осуществляется через соответствующие файлы символьных устройств, которые прикомпонованы к виртуальной файловой системе (VFS).

Символьные драйверы Linux

Рис.8: Общий взгляд на символьный драйвер



Символьные драйверы Linux

- Это означает, что приложение выполняет обычные файловые операции с файлом символьного устройства. Эти операции будут перетранслированы виртуальной файловой системой VFS в соответствующие функции в прикомпонованном драйвере символьного устройства. Затем для того, чтобы получить нужные результаты, с помощью этих функций осуществляется окончательный низкоуровневый доступ к реальному устройству.
- Обратите внимание, что если приложение выполняет обычные файловые операции, их результат не должен отличаться от обычных случаев. Просто для того, чтобы выполнить эти операции, в драйвере устройства будут использоваться соответствующие функции. Например, операция записи с последующей операцией чтения может, в отличие от работы с обычными файлами, не получить то, что только что было записано в файл символьного устройства.

Символьные драйверы Linux

- Помните, что это обычное явление для файлов устройств. Давайте в качестве примера возьмем файл аудио устройства. То, что мы записываем в него, является аудиоданными, которые мы хотим воспроизвести, скажем, через громкоговоритель. Однако при чтении данных мы получим аудио данные, которые мы записываем, например, через микрофон. Записанные данные не обязательно должны быть теми, которые мы воспроизводили.
- В этом полном подключении из приложения к устройству участвуют следующие четыре основных компонента:
- Приложение
- Файл символьного устройства
- Драйвер символьного устройства
- Символьное устройство

Символьные драйверы Linux

- Приложение подключается к файлу устройства при помощи системного вызова open, открывающего файл устройства.
- Файлы устройств подключаются к драйверу устройства с помощью специального механизма регистрации, что осуществляется драйвером. Драйвер связывается с устройством с помощью специальных низкоуровневых операций, характерных для конкретного устройства. Таким образом, мы формируем полное соединение. При этом, обратите внимание, что файл символьного устройства не является реальным устройством, это просто специальная методика (place-holder) подключения реального устройства.
- ***Старший и младший номера файлов устройств***
- При подключении приложения к файлу устройства используется имя файла устройства. Но при подключении файла устройства к драйверу устройства используется номер файла устройства, а не имя файла.

Символьные драйверы Linux

- В результате приложение пользовательского пространства может использовать для файла устройства любое имя, а в пространстве ядра для связи между файлом устройства и драйвером устройства можно использовать тривиальный механизм индексации. Таким номером файла обычно является пара <major, minor>, то есть старший и младший номера файла устройства.
- Ранее (вплоть до ядра 2.4) каждый старший номер использовался в качестве указания на отдельный драйвер, а младший номер использовался для указания на конкретное подмножество функциональных возможностей драйвера. В ядре 2.6 такое использование номеров не является обязательным; с одним и тем же старшим номером может быть несколько драйверов, но, очевидно, с различными диапазонами младших номеров.

Символьные драйверы Linux

- Стандартные старшие номера обычно резервируются для вполне определенных конкретных драйверов. Например, 4 — для последовательных интерфейсов, 13 - для мышей, 14 — для аудио-устройств и так далее. С помощью следующей команды можно будет выдать список файлов различных символьных устройств, имеющихся в вашей системе:
- `$ ls -l /dev/ | grep "^c"`
- ***Использование чисел <major, minor> в ядре 2.6***
- Тип (определен в заголовке ядра linux/types.h):
- dev_t - содержит старший и младший номера
- Макрос (определен в заголовке ядра linux/kdev_t.h):
- MAJOR(dev_t dev) - из dev извлекается старший номер
- MINOR(dev_t dev) - из dev извлекается младший номер
- MKDEV(int major, int minor) - из старшего и младшего номеров создается dev

Символьные драйверы Linux

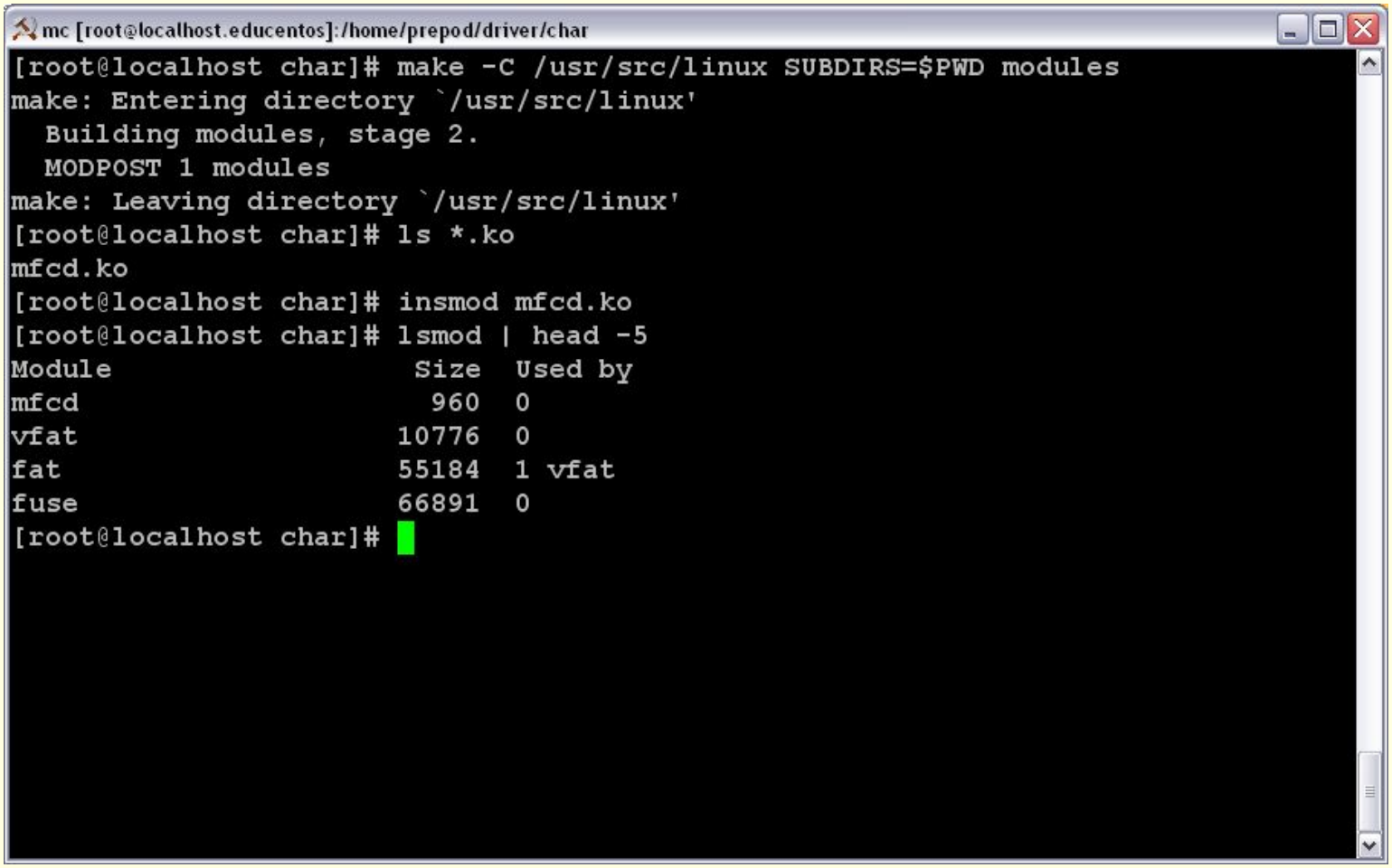
- Подключение файла устройства к драйверу устройства осуществляется за два шага:
- Выполняется регистрация файлов устройств для диапазона <major, minor>
- Подключение операций, выполняемых над файлом устройства, к функциям драйвера устройства.
- Первый шаг выполняется с помощью одного из следующих двух API, определенных в заголовке ядра linux/fs.h:
- int register_chrdev_region(dev_t first, unsigned int cnt, char *name);
- int alloc_chrdev_region(dev_t *first, unsigned int firstminor, unsigned int cnt, char *name);
- С помощью первого API число cnt регистрируется как среди номеров файлов устройств, которые начинаются с first и именем файла name .

Символьные драйверы Linux

- С помощью второго API динамически определяется свободный старший номер и регистрируется число cnt среди номеров файлов устройств, начинающиеся с <the free major, firstminor>, с заданным именем файла name. В любом случае в директории /proc/devices указывается список имен с зарегистрированным старшим номером.
- С учетом этой информации очередная версия нашего драйвера будет иметь следующий вид.
- Повторим обычные шаги, которые узнали при изучении первого драйвера:
- Соберем драйвер (файл .ko), выполнив команду make.
- Загрузим драйвер с помощью команды insmod.
- Выдадим список загруженных модулей с помощью команды lsmod (Рис.9).

Символьные драйверы Linux

Рис.9: Сборка и установка драйвера Linux

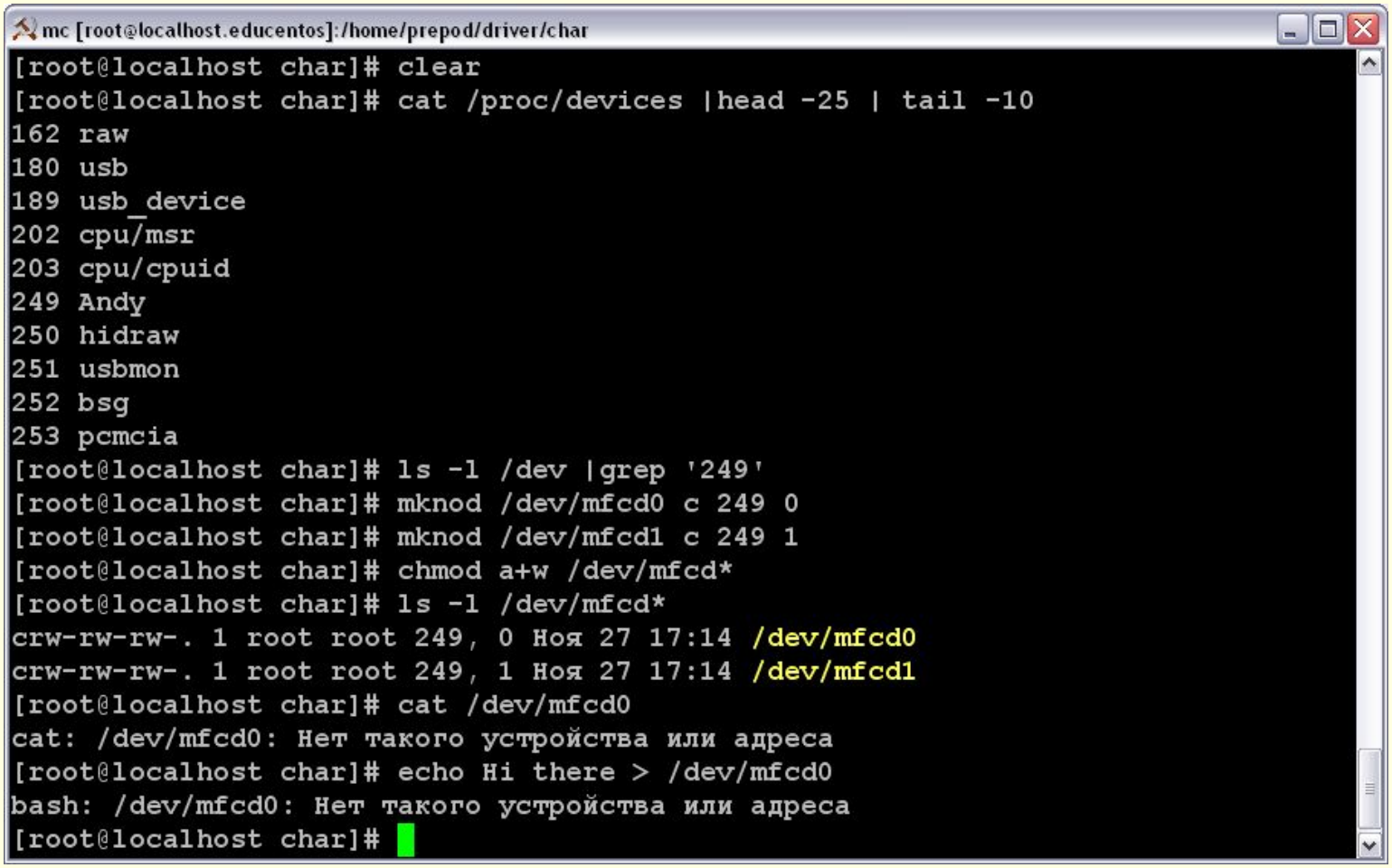


Символьные драйверы Linux

- Перед выгрузкой драйвера с помощью команды rmmod заглянем в директорий /proc/devices для того, чтобы с помощью команды cat /proc/devices найти зарегистрированный старший номер с именем "Andy". Он там был. Тем не менее, мы не сможем в директории /dev найти ни одного файла устройств с таким же старшим номером, т.к. он создан вручную с помощью команды mknod, а затем попытаемся выполнить операции чтения и записи. Все эти действия показаны на рис.10.
- Обратите внимание, что в зависимости от номеров, уже используемых в системе, старший номер 249 может варьироваться от системы к системе. На рис.10 также показаны результаты, которые мы получили при чтении и записи одного из файлов устройств. Очевидно, что все еще не сделан второй шаг подключения файла устройства к драйверу устройства, при котором операции над файлом устройства связываются с функциями драйвера устройства.

Символьные драйверы Linux

Рис.10: Эксперименты с файлом символьного устройства



Файлы символьных устройств

- Даже при регистрации диапазона устройств <major, minor>, файлы устройств в директории /dev не создаются — мы должны были создать их вручную с помощью команды mknod. Но файлы устройств можно создавать автоматически с помощью демона udev. Также необходим второй шаг подключения файла устройства к драйверу устройства — связывание операций над файлом устройства с функциями драйвера устройства.
- ***Автоматическое создание файлов устройств***
- Ранее, в ядре 2.4, автоматическое создание файлов устройств выполнялось самим ядром в devfs с помощью вызова соответствующего API. Однако, по мере того, как ядро развивалось, разработчики ядра поняли, что файлы устройств больше связаны с пользовательским пространством и, следовательно, они должны быть именно там, а не в ядре.

Файлы символьных устройств

- Исходя из этого принципа, теперь для рассматриваемого устройства в ядре в /sys только заполняется соответствующая информация о классе устройства и об устройстве. Затем в пользовательском пространстве эту информацию необходимо проинтерпретировать и выполнить соответствующее действие. В большинстве настольных систем Linux эту информацию собирает демон udev, и создает, соответственно, файлы устройств.
- Демон udev можно с помощью его конфигурационных файлов настроить дополнительно и точно указать имена файлов устройств, права доступа к ним, их типы и т. д. Так что касается драйвера, требуется с помощью API моделей устройств Linux, объявленных в <linux/device.h>, заполнить в /sys соответствующие записи. Все остальное делается с помощью udev. Класс устройства создается следующим образом:

Файлы символьных устройств

- **`struct class *cl = class_create(THIS_MODULE, "<device class name>");`**
- Затем в этот класс информация об устройстве (<major, minor>) заносится следующим образом:
- **`device_create(cl, NULL, first, NULL, "<device name format>", ...);`**
- Здесь, в качестве first указывается dev_t. Соответственно, дополняющими или обратными вызовами, которые должны вызыватся в хронологически обратном порядке, являются:
- **`device_destroy(cl, first);class_destroy(cl);`**
- В случае, если указаны несколько младших номеров minor, API device_create() и device_destroy() могут вызываться в цикле и в этом случае окажется полезной строка <device name format> (<формат имени устройства>).

Файлы символьных устройств

- Например, вызов функции device_create() в цикле с использованием индекса i будет иметь следующий вид:
- **`device_create(cl, NULL, MKNOD(MAJOR(first), MINOR(first) + i), NULL, "mynull%d", i);`**
- ***Операции с файлами***
- Независимо от того, что системные вызовы применяются к обычным файлам, их также можно использовать и с файлами устройств. Если смотреть из пользовательского пространства, то в Linux почти все является файлами. Различие - в пространстве ядра, где виртуальная файловая система (VFS) определяет тип файла и пересылает файловые операции в соответствующий канал, например, в случае обычного файла или директория - в модуль файловой системы, или в соответствующий драйвер устройства в случае использования файла устройства. Мы будем рассматривать второй случай.

Файлы символьных устройств

- Теперь, чтобы VFS передала операции над файлом устройства в драйвер, ее следует об этом проинформировать. Это называется регистрацией драйвером в VFS файловых операций. Регистрация состоит из двух этапов.

1. Занесем нужные нам файловые операции (my_open, my_close, my_read, my_write, …) в структуру, описывающую файловые операции (struct file_operations pugs_fops) и ею инициализируем структуру, описывающую символьное устройство (struct cdev c_dev); используем для этого обращение cdev_init().
2. Передадим эту структуру в VFS с помощью вызова cdev_add(). Обе операции cdev_init() и cdev_add() объявлены в <linux/cdev.h>. Естественно, что также надо закодировать фактические операции с файлами (my_open, my_close, my_read, my_write).

Файлы символьных устройств

- Теперь, чтобы VFS передала операции над файлом устройства в драйвер, ее следует об этом проинформировать. Это называется регистрацией драйвером в VFS файловых операций. Регистрация состоит из двух этапов.

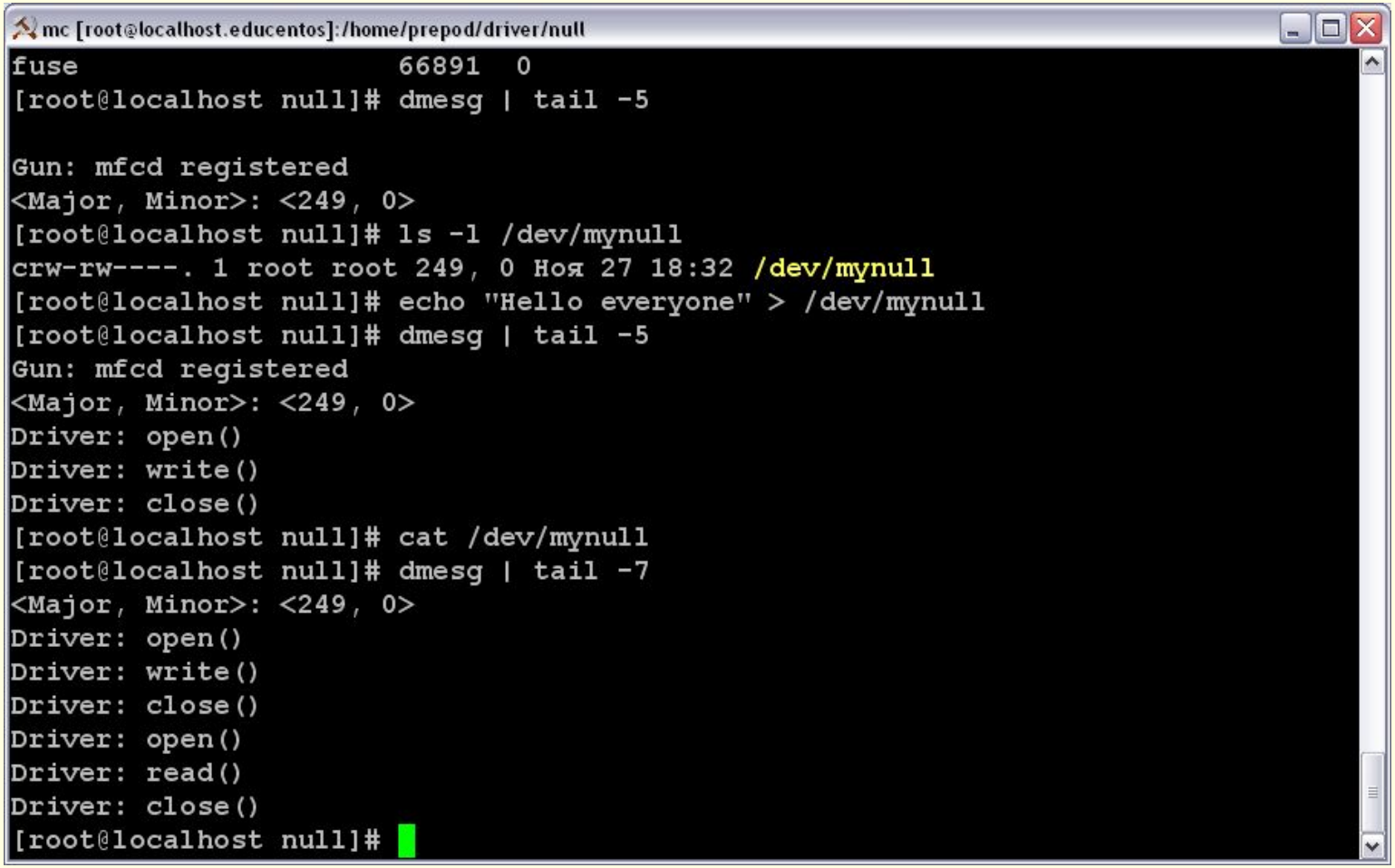
1. Занесем нужные нам файловые операции (my_open, my_close, my_read, my_write, …) в структуру, описывающую файловые операции (struct file_operations pugs_fops) и ею инициализируем структуру, описывающую символьное устройство (struct cdev c_dev); используем для этого обращение cdev_init().
2. Передадим эту структуру в VFS с помощью вызова cdev_add(). Обе операции cdev_init() и cdev_add() объявлены в <linux/cdev.h>. Естественно, что также надо закодировать фактические операции с файлами (my_open, my_close, my_read, my_write).

Файлы символьных устройств

- Итак, для начала, давайте все это сделаем как можно проще - скажем, максимально просто в виде "null драйвера".
- Повторим обычный процесс сборки, добавив при этом некоторые новые проверочные шаги, а именно:
- Соберем драйвер (файл .ko) с помощью запуска команды make.
- Загрузим драйвер с помощью команды insmod.
- С помощью команды lsmod получим список всех загруженных модулей.
- С помощью команды cat /proc/devices. получим список используемых старших номеров major.
- Поэкспериментируем с "null драйвером" (подробности смотрите на рис.11).
- Выгрузим драйвер с помощью команды rmmod.

Файлы символьных устройств

Рис.11: Эксперименты с "null драйвером"



Файлы символьных устройств

- В своем драйвере мы использовали свои собственные вызовы (my_open, my_close, my_read, my_write), но они, в отличие от любых других вызовов файловой системы, работают таким необычным образом: что бы мы не записывали, при чтении мы ничего не можем получить.
- Заметим, что возврат значения из функций my_open() и my_close() тривиален, типы возвращаемых значений - int, и обе функции возвращают нулевое значение, что означает успешное завершение.
- Но типы возвращаемых значений обоих функций my_read() и my_write() не int, а - ssize_t. При дальнейшем исследовании заголовков ядра, оказалось, что возвращаемое значение должно быть словом со знаком. Итак, если возвращается отрицательное число , то обычно это ошибка. Но неотрицательное возвращаемое значение будет иметь дополнительный смысл.

Файлы символьных устройств

- Для операции чтения, оно будет указывать количество читаемых байтов, а для операции записи, оно будет указывать количество записываемых байтов.
- ***Чтение файла устройства***
- Когда пользователь выполняет чтение из файла устройства /dev/mynull, этот системный вызов поступает в слой виртуальной файловой системы (VFS), находящийся в ядре. VFS декодирует пару <major, minor> и выясняет, что нужно перенаправить системный вызов в функцию драйвера my_read(), которая зарегистрирована в виртуальной системе. Так что с этой точки зрения функция my_read() вызывается у нас, писателей драйверов устройств, как запрос на чтение. И, следовательно, возвращаемое значение будет указывать лицу, сделавшему запрос (например, пользователю), сколько байтов они получают при запросе на чтение.

Файлы символьных устройств

- В нашем примере null-драйвера мы возвратили ноль - это означает, что доступных байтов данных нет или что, другими словами, был достигнут конец файла. И, следовательно, когда читается файл устройства, то независимо от того, что в него было записано, результат будет отсутствовать.
- На самом деле, функция my_read() должна записать данные в буфер buf (переменная — буфер, которая является вторым параметром функции my_read() и указывается пользователем) в соответствие со значением len (третий параметр функции), количеством байтов, запрашиваемых пользователем.
- Если более конкретно, число байтов, записываемых в буфер buf, должно быть меньше или равно значению len, а количество записанных байтов должно быть передано обратно в качестве возвращаемого значения.

Файлы символьных устройств

- Нет, это не опечатка - в операции чтения писатели драйверов устройств "записывают" данные в буфер, который предоставляется пользователем. Мы (возможно) читаем данные из соответствующего устройства, а затем записываем эти данные в пользовательский буфер, так что пользователь может его прочитать.
- ***Запись в файл устройства***
- Операция записи действует наоборот. Пользователь предоставляет значение длины len (третий параметр функции my_write()), указывающий количество байтов данных, которые должны быть записаны и которые расположены в буфере buf (второй параметр функции my_write()). Функция my_write() будет читать эти данные и, возможно, записывать их на соответствующее устройство, и возвратит число, равное количеству байтов, которые были успешно записаны.

Файлы символьных устройств

- Изменим my_read() и my_write() следующим образом, добавив статический глобальный символьный массив.
- Так как драйвер работает в пространстве ядра, то он отграничен от адресного пространства пользователя. А нам хотелось бы иметь возможность вернуть некий результат. Для этого используется функция put_user(). Она как раз и занимается тем, что перекидывает данные из пространства ядра в пользовательское. Наоборот действует функция get_user(). Обе они прописаны в файле включения <asm/uaccess.h>.
- Теперь действия по записи данных в устройство и чтения из него приведут к результату, показанному на рис. 12.
- Все, что нам осталось – это написать пользовательское приложение, работающее с нашим драйвером.

Файлы символьных устройств

Рис.12: Работа с "null драйвером"